ETSI TR 102 026 V1.1.1 (2002-01)

Technical Report

Telecommunications and Internet Protocol Harmonization Over Networks (TIPHON); Study of the use of TTCN-3 for SIP and for OSP test specifications

Reference

DTR/TIPHON-06019

Keywords

VoIP, interoperability, IP, testing, TTCN

Contents

Intellectual Property Rights

Foreword

This Technical Report (TR) has been produced by ETSI Project Telecommunications and Internet Protocol Harmonization Over Networks (TIPHON).

1 Scope

The present document provides an analysis on the suitability of using TTCN-3 as defined in ES 210 873-1 [1] to specify the test specifications for TIPHON protocols, in particular the TIPHON profile of SIP (Session Initiation Protocol) and the TIPHON OSP (Open Settlement Protocol). This study is restricted to the use of the TTCN-3 Core Language.

2 References

For the purposes of this Technical Report (TR) the following references apply:

[1] ETSI ES 201 873-1: "Methods for Testing and Specification (MTS); The Tree and Tabular Combined Notation version 3; Part 1: TTCN-3 Core Language".

[2] ISO/IEC 9646-3: "Information technology - Open Systems Interconnection - Conformance testing methodology and framework - Part 3: The Tree and Tabular Combined Notation (TTCN) Edition 2".

[3] ETSI TS 101 321: "Telecommunications and Internet Protocol Harmonization Over Networks (TIPHON); Open Settlement Protocol (OSP) for Inter-Domain pricing, authorization, and usage exchange".

[4] ITU-T Recommendation Z.140: "The tree and tabular combined notation version 3 (TTCN-3): Core language".

3 Abbreviations

For the purposes of the present document, the following abbreviations apply:

ASN.1	Abstract Syntax Notation One
ATS	Abstract Test Suite
DTD	Document Type Definition
IUT	Implementation Under Test
MTC	Master Test Component
OSP	Open Settlement Protocol
(P)ICS	(Protocol) Implementation Conformance Statement
(P)IXIT	(Protocol) Implementation eXtra Information for Testing
PDU	Protocol Data Unit
SIP	Session Initiation Protocol
SUT	System Under Test
TCP	Transfert Control Protocol
TTCN-2	Tree and Tabular Combined Notation version 2
TTCN-3	Testing and Test Control Notation version 3
UDP	User Datagram Protocol
XML	eXtensible Markup Language
PCO	Point of Control and Observation
DE	Development Environment

4 Background

The detailed code for nearly all ETSI (conformance) Abstract Test Suites (ATS) is written in TTCN. There are two versions of TTCN, version 2 (TTCN-2) as defined in ISO/IEC-9646-3 [2] and the recently published ETSI version 3 ES 201 873-1 [1].

NOTE: Version 1 of TTCN is not now used by ETSI.

Version 2 is oriented towards conformance testing and has been widely applied in testing telecommunications protocols and services for over 10 years. TTCN-3 is a modernization of TTCN-2. It has been developed to apply to a wide range of testing applications (i.e. it is not limited to conformance testing) and the syntax of the language has been brought into line with that of other modern programming languages.

While it is not anticipated that TTCN-2 will immediately replace TTCN-3 (from ETSI's point of view the transition to TTCN-3 is expected to occur over several years) there are good reasons to consider using TTCN-3 for "new" protocols such as SIP or OSP.

EP TIPHON is writing test specifications for H.225, H.245, H.248, SIP and OSP. The tests for the first three protocols are being written in TTCN-2. This is mainly due to timing (the work was started several months prior to the publication of TTCN-3) and the fact that they are "traditional" protocols (for example H.225 is very close to Q.931). It is also more likely that, in the short-term, the actual test systems for these protocols will be based on TTCN-2.

However, the nature of SIP and OSP (e.g., text-based, datacom-oriented) makes them an ideal candidate for TTCN-3. The present document makes an initial analysis on the suitability of using TTCN-3 to for SIP and OSP test specifications.

5 Suitability of TTCN-3 for SIP testing

In order to understand the suitability of TTCN-3 for testing SIP it is necessary to consider three main aspects:

- the basic testing architecture, i.e. the location of the test interfaces;
- the expression of dynamic behaviour (i.e. SIP message exchanges);
- the representation of data (i.e. SIP messages).

These aspects are described in clauses 5.1, 5.2 and 5.3 respectively.

5.1 Architectural considerations for testing SIP

Two conceptual SIP test systems are illustrated in figure 1. The TTCN-3 parts of the test system are represented by the white boxes, which in the present document we refer to as the "TTCN-3 Tester". The light grey box represents sub-structured parts of the test system. The dark grey boxes indicate the underlying transport layer, either UDP or TCP.

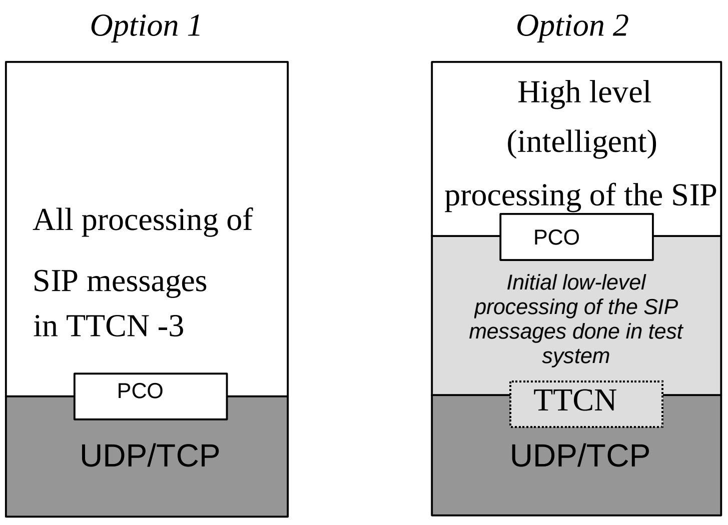
Option 1
Option 2
All processing of SIP messages in TTCN -3
PCO
UDP/TCP
High level (intelligent) processing of the SIP
PCO
Initial low-level processing of the SIP messages done in test system
TTCN
UDP/TCP

Figure 1: Basic test system architecture

TTCN-3 behaviour is executed over test ports, sometimes called PCOs (Points of Control and Observation). For SIP testing there are basically two options for the placement of the PCO.

- directly over UDP (or TCP);
- higher than UDP (or TCP), i.e. "embedded" in the test system.

In the first option of figure 1 all processing of the SIP messages is expressed in TTCN-3. For received messages this means that the PCO delivers a SIP message to the TTCN-3 tester as a single text string. The TTCN-3 code must then (somehow) parse this text string and break it down into data structures on which the TTCN-3 matching mechanisms etc. can operate. For send messages the reverse occurs i.e. TTCN-3 data structures representing the SIP message are encoded as a single text string.

It is certainly possible to use TTCN-3 this way but this would probably be inefficient. It would also overload the TTCN-3 test cases (not to mention the test suite writers) with detail not explicitly relevant to the test purposes.

In the second option, the test system receives a SIP message over the UDP (or TCP) port and does the initial parsing *before* passing the structure to the TTCN-3 Tester via the PCO. In its simplest form this parser need only recognize the basic "outline" of the message with no detailed knowledge of individual headers. This structure would be mapped to the corresponding TTCN-3 template on a best possible fit basis. The TTCN-3 Tester then operates directly on this data structure rather than the incoming text string (by pattern matching).

If the tester is to deliver more complex TTCN-3 structures then the underlying parser will need to be correspondingly complex. As this will effect how a TTCN-3 test case is expressed (i.e. place restrictions on how TTCN-3 is used) it is important that this functionality is defined by EP TIPHON at an early stage.

In conventional protocol testing (especially when using, say, ASN.1) this sub-layer (shaded light grey in figure 1) is often referred to as an encoder-decoder. Here, the incoming data is a bit stream which is decoded by the test system and passed to the TTCN-3 tester in structured form.

Discussions with several tool implementer's indicate that option 2 should be the favoured approach. The present document therefore recommends that EP TIPHON follow option 2 when writing TTCN-3 test cases for SIP.

5.2 Expressing SIP dynamic behaviour in TTCN-3

SIP has very simple dynamic behaviour. The TTCN-3 communication and timer mechanisms etc. are entirely adequate to specify the exchange of SIP messages. The present document recommends that TIPHON SIP tests are expressed using asynchronous communication.

NOTE: Generally, SIP testing would be based on asynchronous message exchanges, however TTCN-3 does have synchronous communication if it is desired to express the test that way.

A typical piece of SIP behaviour could be:

```
testcase SIP_RG_RT_V_001() runs on SipComponent system SipInterfaces
    // Selection: To be defined
    // Status: Mandatory
    // SUT: A UA, a proxy, or a redirect server.
    // Precondition: None
    // Ref: 2.2 [1], 7.1 [1], 10.14 [1]
    // Purpose: Ensure that the IUT, in order to be registered, sends a REGISTER request
    // t to its proxy (Home server, outbound proxy) with the action field set to "proxy"
    // in the Contact header field, without user name in the Request-URI,
    // with a Via header field and with a SIP URL as request-URI.
    {
        var REG_Request V_REGISTER_Request;
        var ContactAddress_List V_ContactList;
        var GenericParam_List V_GenericParamList;
        var integer i,j, nbelement, nbparam;
        var boolean hasBeenFound:=false;

        sut.action ("Please REGISTER");
        TWait.start(PX_TWAIT);
        alt
        {
        [] SIP1.receive (REGISTER_Request_r_2) from rcv_label -> value V_REGISTER_Request
          {
            TWait.stop;
            // Catch and prepare informations to answer
            iutContact :=
      getContactAddr(V_REGISTER_Request.reqHeader.contact.contactBody.contactAddress_List[1]);

            V_CallId := V_REGISTER_Request.reqHeader.callId;
            V_CSeq := V_REGISTER_Request.reqHeader.cSeq;
            V_From := V_REGISTER_Request.reqHeader.fromField;
            V_To := V_REGISTER_Request.reqHeader.toField;
            V_Via := V_REGISTER_Request.reqHeader.via;
            // update sent_label according to received via header field
            getViaReplyAddr(V_Via.viaBody);
            //Add a Tag in the TO field
            V_To.toParams := {{TAG_ID, GetAValueTag()}};

            // Check Contact content
            V_ContactList := V_REGISTER_Request.reqHeader.contact.contactBody.contactAddress_List;
            nbelement := sizeof(V_ContactList);
                for (i:=1;i==nbelement;i:=i+1)
                {
                  hasBeenFound:=false;
                  // Check that parameters are present in the contact
                    if (match(V_ContactList[i], ContactAdress_r_1))
                    {
                        V_GenericParamList := V_ContactList[i].contactParams;
                        nbparam := sizeof(V_GenericParamList);
                        j:=1;
                        //Check that at least one parameter is set to action="proxy"
                        do
                        {
                            if (match(V_GenericParamList[j],GenericParam_r_1))
                            {
                                hasBeenFound:=true;
                            }

                        // Check that contact does not include a parameter set to action="redirect"
                            if (match(V_GenericParamList[j],GenericParam_r_2))
                            {
                                hasBeenFound:=false;
                                j:= nbparam;
                            }
                            j:=j+1;
                        }
                        while (j<=nbparam) //end loop on contact parameters
                    }

                    if(not hasBeenFound)
                    {
                      verdict.set(fail);
                      //Answer with a 409 status message
                      SIP1.send (Response_409_s_1(V_CallId, V_CSeq, V_From, V_To,
                      V_Via )) to sent_label; stop
                    }
                } //end For on Contact list
                verdict.set(pass);
            //Send a 200OK Answer to the UA with an Expire header field set
            //to PX_DELTA_REGISTRATION and the contact list
            SIP1.send (Response_200_s_2(V_CallId, V_CSeq, V_From, V_To,
             V_Via, V_REGISTER_Request.reqHeader.contact,
              PX_DELTA_REGISTRATION )) to sent_label


        }
      [] SIP1.trigger from rcv_label
      {
          all timer.stop;
          verdict.set(fail);
          stop
      }
      [] Twait.timeout  {verdict.set(inconc); stop}
     }

    } // end testcase SIP_RG_RT_V_001
```

5.3 Expressing SIP messages in TTCN-3

Currently many SIP test suites specify one single text string for each instance of a message. Changing one element in a message means that a complete, new message needs to written. The end result is many hundreds of individual SIP messages. No rationalization. No reuse. Worse still, matches on incoming messages have to be exact, where in practice a degree of flexibility is often desirable.

The TTCN-3 approach allows to set and match individual elements of data in complex messages. To give a high degree of controllability and observability. Because SIP messages are text based they have no explicit structure, contrary to conventional telecommunications protocol. For example:

```
INVITE sip:test@sip.com SIP/2.1
From: userB<sip:xxx@yyy.zzz>
To: userA<sip:aaa@bbb.ccc>
CSeq: 1 INVITE
Content-Length: 0
```

In order to test SIP using TTCN-3 it is necessary to give the messages at least some level of structuring. Highly structured SIP data will give a good degree of control but will probably lead to a humanly unreadable test suite. Conversely, little or no structuring will give good readability but very little control. In this clause we present a style of using TTCN-3 that attempts to achieve controllability while retaining a good degree of readability.

A SIP message has three basic parts, the *Request (or Status) line*, the *headers* and the (optional) message *body*. The components of a Request or Status line appear in a given order. In TTCN-3 this can be represented using a record type, for example:

```
// SIP Message Request
type record SIP_REQUEST
{   charstring      Method          optional, // even mandatory fields are optional
    charstring      Request_URI     optional, // so that we can specify invalid messages
    charstring      SIP_Version     optional,
    :
}
```

Actual messages can be defined using TTCN-3 templates. For example:

```
template SIP_REQUEST MyRequest :=
{   Method      := "INVITE ",
    Request_URI := "sip:test@sip.com ",
    SIP_Version := "SIP/2.1\r\n"  // where \r\n represents %d13%d10 the CR + LF characters
    :
}
```

Explicit spaces could be included in the structure rather than having them as part of the actual string value
(see clause 5.3.1).

For the sake of this discussion let us assume that SIP headers are text strings terminated by an end of line character (e.g., CR or LF or CRLF). Generally, SIP messages allow headers to appear in any order. However, for sent messages (i.e. SIP Requests) the TTCN-3 Tester should specify messages with the SIP headers in a given order. In TTCN-3 this can be expressed using the **record of** type.

```
// Unbounded array of character strings (i.e. headers)
type record of charstring REQUEST_HEADERS;

// Unbounded array of character strings (i.e. body elements)
type record of charstring REQUEST_BODY;

// SIP Message = Request Line + Headers + Body
type record SIP_REQUEST
{   charstring      Method          optional, // even mandatory fields are optional
    charstring      Request_URI     optional, // so that we can specify invalid messages
    charstring      SIP_Version     optional,
    REQUEST_HEADERS Message_Headers optional,
    REQUEST_BODY    Message_Body    optional
}
```

For received messages (i.e. SIP Responses) the TTCN-3 Tester should be prepared to accept messages with the SIP headers appearing in an arbitrary order. In TTCN-3 this can be expressed using the **set of** type.

```
// Unbounded set of character strings (i.e. headers)
type set of charstring RESPONSE_HEADERS;

// Unbounded set of character strings (i.e. body elements)
type set of charstring RESPONSE_BODY;

// SIP Message = Response Line + Headers + Body
type record SIP_RESPONSE
{   charstring          SIP_Version     optional, // even mandatory fields are optional
    charstring          Status_Code     optional, // so that we can specify invalid messages
    charstring          Reason_Phrase   optional,
    RESPONSE_HEADERS    Message_Headers optional,
    RESPONSE_BODY       Message_Body    optional
}
```

5.3.1 SIP headers

In its simplest form a SIP header can be represented as a single text string, as described in clause 5.3. For example:

```
"CSeq: 1 INVITE\r\n"
```

However, the CSeq header could be represented as a structured type of several elements:

```
    type record CSEQ
    {   tag     charstring,
        sp1     charstring,
        counter charstring,
        sp2     charstring,
        method  charstring  // could also be enumerated type
    }

    //  with the value:
    template CSEQ MyCSeq :=
    {   tag     := "CSeq:",
        sp1     := " ",
        counter := "1",
        sp2     := " ",
        method  := "INVITE",
        eol     := "\r\n"
    }
```

Representing headers in this way gives a wide range of control (we can explicitly access individual fields in SIP headers). However, readability can be seriously affected if this approach is not used care. Note also that the REQUEST_HEADERS type (for example) can no longer be a simple **record of charstring** but would need, for example, to be a **union** of the different message header structures. See also clause 3.3.2.2.

One method to give some form of pseudo-structure to the **charstring** and yet retain readability is to use the concatenator operator, for example (where CRLF = "\r\n" and SP = " "):

```
    "CSeq:" & SP & "1" & SP & "INVITE" & CRLF
```

NOTE: Regular expressions (patterns) will be introduced into TTCN-3 see clause 5.3.2 for a description and clause 5.3.2.3 for a SIP example.

```
  template SIP_RESPONSE MyResponse :=
  { SIP_Version     := "INVITE" & SP,
    Status_Code     := "200" & SP,
    Reason_Phrase   := "OK" & CRLF,

    Message_Headers := {    "From:" & "userB<sip:xxx@yyy.zzz>" & CRLF,
                            "To:" & "userA<sip:aaa@bbb.ccc>" & CRLF,
                            "CSeq:" & "1" & "INVITE" & CRLF,
                            "Content-Length: " & "0",
                            *   // All other headers are ignored
                        },
    Message_Body   := omit
  }
```

This approach makes more sense when used with parameterized templates and wildcards, as explained in clauses 5.3.1.1 and 5.3.1.2.

See annex A for a full example (test case) which applies the various approaches described in the previous clauses.

5.3.1.1 Parameterization

TTCN-3 templates can be parameterized. For example, we could parameterize the counter in the CSeq header:

```
    template SIP_RESPONSE MyResponse (charstring COUNTER) :=
    {   :
        Message_Headers :=  {   :
                                "CSeq:" & COUNTER & "INVITE" & CRLF
                                :
                            },
        Message_Body    := omit
    }

    // And in the receive line we could write:
    pco.receive(MyRespons("1"))
```

5.3.1.2 Wildcards

The TTCN-3 matching mechanisms (wildcards) can also be used. For example, we could wildcard the counter in the CSeq header:

```
    template SIP_RESPONSE MyResponse :=
    {   :
        Message_Headers :=  {   :
                            "CSeq:" & <?> & "INVITE" & CRLF
                            },
        Message_Body    := omit
  }
```

NOTE: The exact syntax of TTCN-3 wildcards and patterns is still under review.

5.3.1.3 Using modified templates

Another mechanism that could be used is derived, or modified, templates. For example, a complete SIP message would contain all the headers, say, from "Allow" to "WWWAuthenticate".

```
    // SIP Message Request
    type record SIP_REQUEST
    {   charstring      Method          optional, // even mandatory fields are optional
        charstring      Request_URI     optional, // so that we can specify invalid messages
        charstring      SIP_Version     optional,

        charstring      Allow           optional,
            :
        charstring      From            optional,
        charstring      To              optional,
        charstring      CSeq            optional,
        charstring      Content-Length  optional
            :
        charstring      WWWAuthenticate optional,

        RESPONSE_BODY   Message_Body    optional
    }
```

A base response template where all headers are omitted:

NOTE: For SIP Responses the wildcard "*" would probably be used instead of **omit**.

```
  template SIP_REQUEST MyBaseRequest :=
  { Method      := "INVITE" & SP,
    Request_URI := "sip:test@sip.com" & SP,
    SIP_Version := "SIP/2.1" & CRLF,

    Allow           :=  omit,
        :
    From            :=  omit,
    To              :=  omit,
    CSeq            :=  omit,
    ContentLength   :=  omit,
        :
    WWWAuthenticate :=  omit,

    MessageBody     :=  omit
  }
```

The following template produces the same SIP message as the example in clause 5.3. That is, by default all other headers are omitted.

```
  template SIP_REQUEST MyRequest modifies SIP_REQUEST MyBaseRequest :=
  { Method      := "INVITE" & SP,
    Request_URI := "sip:test@sip.com" & SP,
    SIP_Version := "SIP/2.1" & CRLF,

    From            :=  "From:" & "userB<sip:xxx@yyy.zzz>" & CRLF,
    To              :=  "To:" & "userA<sip:aaa@bbb.ccc>" & CRLF,
    CSeq            :=  "CSeq:" & "1" & "INVITE" & CRLF,
    ContentLength   :=  "Content-Length: " & "0"
  }
```

5.3.2 TTCN-3 regular expressions

TTCN-3 now supports limited regular expressions, or patterns. These may be used to match character string values anywhere that wildcards and matching mechanisms are currently allowed in TTCN-3.

Work is progressing to introduce more complicated pattern-matching into the language. It is expected that tool-makers will implement these proposals as they develop. This will ensure that, for EP TIPHON at least, pattern matching capabilities will be available in a timely manner.

5.3.2.1 Simple patterns

In addition to literal characters, character patterns allow the use of meta characters "?" and "*" to mean any character and any number of any character respectively. For example:

```
template charstring MyTemplate:= pattern "ab??xyz*";
```

This template would match any character string that consists of the characters "ab", followed by any two characters, followed by the characters "xyz", followed by any number of any characters.

The metacharacter "\" is used as an escape character. For example:

```
template charstring MyTemplate:= pattern "ab?\?xyz*";
```

This template would match any character string which consists of the characters "abr", followed by any characters, followed by the characters "?xyz", followed by any number of any characters.

In addition to direct string values it is also possible within the pattern statement to use references to existing templates, constants or variables. The reference shall resolve to one of the character string types and more than one. For example:

```
const charstring MyString:= "ab?";

template charstring MyTemplate:= pattern MyString;
```

This template would match any character string that consists of the characters "ab", followed by any characters. In effect any character string following the **`pattern`** keyword either explicitly or by reference will be interpreted following the rules defined in this clause.

5.3.2.2 More complex patterns

The draft proposal (July 2001) for a more sophisticated pattern matching mechanism in TTCN-3 is described below. This proposal (or something very similar) will be incorporated into the language by October 2001 (support in tools for this feature is already available).

The list of meta characters for TTCN-3 patterns is shown in table 1.

Table 1: List of TTCN-3 Pattern Metacharacters

Metacharacter	Description
?	Match any character
*	Match any character zero or more times
\	Cause the following meta character to be interpreted as a literal
[ ]	Match any character within the specified set
{ *group, plane, row, cell* }	Match the Universal character specified by the quadruple
<*reference*>	Insert the referenced user defined string and interpret it as a regular expression
\d	Match any numerical digit (equivalent to [0-9])
\w	Match any alphanumeric character (equivalent to [0-9a-zA-Z])
\\	Match the backspace character
""	Match the double quote character
|	Used to denote two alternative expressions
( )	Used to group an expression
#(n, m)	Match the preceding expression at least n times but no more than m times

5.3.2.2.1 Set expression

The set expression is delimited by the "[" "]" symbols. In addition to character literals, it is possible to specify character ranges using the separator "-". The set expression can also be negated by placing the "^" character as the first character after the opening square brace.

For example:

```
template charstring RegExp1:= pattern “[a-z]”;  // this will match any character from a to z

template charstring RegExp2:= pattern “[^a-z]”;  // this will match any character except a to z

template charstring RegExp3:= pattern “[A-E][0-9][0-9][0-9]YKE”;

// RegExp3 will match a string which starts with a letter between A and E then has three
// digits and the letters YKE
```

5.3.2.2.2 Reference expression

In addition to direct string values it is also possible within the pattern statement to use references to existing templates, constants or variables. The reference is enclosed within the "<" ">" characters. The reference shall resolve to one of the character string types. For example:

```
const charstring MyString:= "ab?";

template charstring MyTemplate:= pattern “<MyString>”;
```

This template would match any character string that consists of the characters "ab", followed by any characters. In effect any character string following the **`pattern`** keyword either explicitly or by reference will be interpreted following the rules defined in this clause.

```
template universal charstring MyTemplate1:= pattern “<MyString>de{1, 1, 13, 7}”;
```

This template would match any character string which consists of the characters "ab", followed by any characters, followed by the characters "de", followed by the character in ISO10646 with group = 1, plane = 1, row = 13 and cell = 7.

5.3.2.2.3 Match expression n times

To specify that the preceding expression should be matched a number of times the "#(n, m)" syntax is used. This specifies that the preceding expression must be matched at least n times but not more than m times.

For example:

```
template charstring RegExp4:= pattern “[a-z]#(9, 11)”;   // match at least 9 but no more than 11
                                                         // characters from a to z
template charstring RegExp5:= pattern “[a-z]#(9)”;       // match exactly 9
                                                         // characters from a to z
template charstring RegExp6:= pattern “[a-z]#(9, )”;     // match at least 9
                                                         // characters from a to z
template charstring RegExp7:= pattern “[a-z]#(, 11)”;    // match no more than 11
                                                         // characters from a to z
```

5.3.2.3 Using regular expressions with SIP

Patterns would replace the syntax illustrated in clause 5.1. For example, the following pattern indicates that "CSeq" may be followed by 1 or more spaces, that the counter value (e.g., "1") is in the template parameter COUNTER (see clause 5.3.1.1) and that "INVITE" is preceded by at least one space and followed by exactly one CRLF.

```
pattern "CSeq:<SP>#(1,)<COUNTER><SP>#(1,)INVITE<CRLF>"
```

Following this approach the example of 5.3.1 would become:

```
template SIP_REQUEST MyRequest (charstring COUNTER) :=
{ SIP_Version     := pattern "INVITE<SP>#(1,)",
  Status_Code     := pattern "200<SP>#(1,)",
  Reason_Phrase   := pattern "OK<CRLF>",

  From            :=  pattern "From:<SP>#(1,)\<userB<sip:xxx@yyy.zzz\><CRLF>",
  To              :=  pattern "To: :<SP>#(1,)\<userA<sip:aaa@bbb.ccc\><CRLF>",
  CSeq            :=  pattern "CSeq:<SP>#(1,)<COUNTER><SP>#(1,)INVITE<CRLF>",
  Content-Length  :=  pattern "Content-Length:<SP>#(1,)0<CRLF>"
}
```

6 Suitability of TTCN-3 for OSP testing

In order to understand the suitability of TTCN-3 for testing OSP it is necessary to consider three main aspects:

- the basic testing architecture, i.e. the location of the test interfaces;
- the expression of dynamic behaviour (i.e. OSP message exchanges);
- the representation of data (i.e. OSP messages).

These aspects are described in clauses 6.1, 6.2 and 6.3 respectively.

6.1 Architectural considerations for testing OSP

For the test architecture the discussion of clause 5.1 also applies to OSP. A more detailed test architecture is shown in figure 2. The adaptation layer is shown in more detail in figure 3.

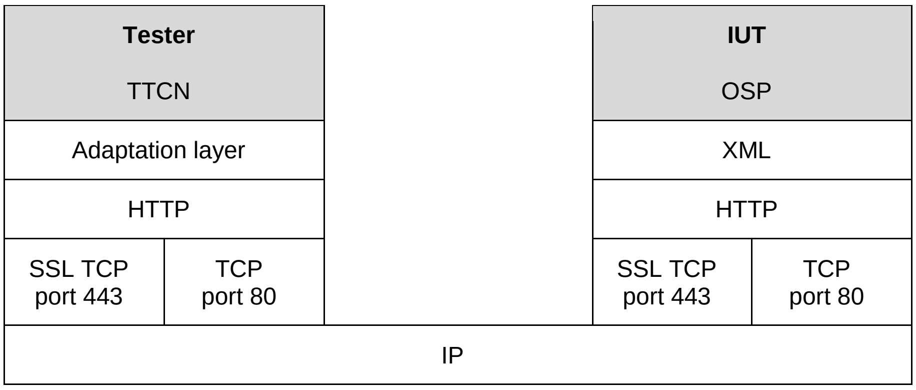

Figure 2: Test architecture

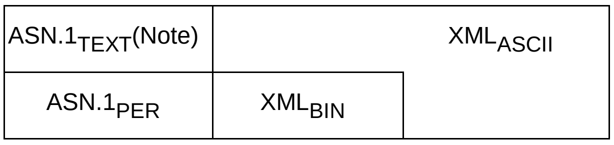

Figure 3: Adaptation layer

6.1.1 Normal OSP message exchange

In the case of normal OSP message exchange the adaptation layer shall proceed the following tasks:

- The Adaptation layer shall receive an ASCII string (XML_{ASCII}) or a binary presentation of the XML document (XML_{BIN}) from the HTTP layer.
- The Adaptation layer shall decode XML_{ASCII} or XML_{BIN} according to the OSP DTD. The OSP DTD is defined in TS 101 321 [3], annex A.
- The Adaptation layer shall map the decoded result to the TTCN3 types. The TTCN3 types are defined in annex C.
- The Adaptation layer shall add all the raw data received from the HTTP layer in the payload field of the TTCN3 type. If a matching error in the Tester occurs, the raw data will be compared with the parsed data in order to see if the matching error is a result of an OSP protocol error or if it is a result of a parsing error.

6.1.2 Token carriage

In the case of Token Carriage the Adaptation layer may receive the following formats:

- ASN.1_{PER}: The Adaptation layer receives an ASN.1_{PER} format as defined in TS 101 321 [3], clause D.2.1. The Adaptation layer shall convert the ASN.1_{PER} format to the TTCN3 types. The TTCN3 types are defined in annex C of [3].
- XML_{ASCII}: The Adaptation layer receives an ASCII string as defined in TS 101 321 [3], clause D.2.2. The rules of clause 4.2.1 shall apply.
- XML_{BIN} : The Adaptation layer receives a binary presentation of the XML document as defined in TS 101 321 [3], clause D.2.3. The rules of clause 4.2.1 shall apply.

6.2 Expressing OSP dynamic behaviour in TTCN-3

The dynamic behaviour of OSP is trivial, comprising simple Request/Response (Client/Server) interchanges.

```
testcase    OSP_SV_PRI_BV_001() runs on MTC_OSP
    // Selection:   Pics Table I.1/2 [1] AND Pics Table I.20/1 [1] AND Pics Table I.20/2
    // [1]
    // Precondition:    None
    // Ref: Clause 6.2.1 [1]
    // Purpose: Ensure that the IUT, on receipt of a PricingIndication,
    // sends a PricingConfirmation with a componentID attribute associated to the Prici
    // ngIndication, with a Timestamp and a Status element.
    // The Timestamp element shall contain the definition of the time according to ISO
    // 8601 [8]. The Status element shall contain the Code element. The Code element sh
    // all contain the code value 2xx.
    {
        //activate DF_Server();

        var integer TCV_MsgId;
        var integer TCV_CompId;
        var integer TCV_Code;
        var integer TCV_Result[3];

        var PricingConfirmation TCV_PriConf;

        TCV_Result:= Calc_Ids();
        TCV_MsgId:= TCV_Result[0];
        TCV_CompId:= TCV_Result[1];

        OSP_Init(IUTisServer_E, PX_TCPPort);

        L1.send(PriInd_S1(TCV_MsgId, TCV_CompId, PX_Currency, PX_Amount,
                                    PX_Increment,PX_Unit));
        TAC.start;
        alt
        {
            []  L1.receive(PriConf_R1(TCV_MsgId, TCV_CompId)) -> value TCV_PriConf
                    {
                        TCV_Code:= TCV_PriConf.pricingConfirmationContents.
                        status.statusContents.code;
                        TAC.stop
                    }
            []  TAC.timeout
                    {
                        verdict.set(fail);
                        stop
                    }
        }

        if  ((200<= TCV_Code)and (TCV_Code<= 299) )
                {verdict.set(pass)}
        else
                {verdict.set(fail)}
    } // end testcase   OSP_SV_PRI_BV_001
```

6.3 Expressing OSP messages in TTCN-3

OSP is XML-based and the real complexity of the protocol is in the OSP messages.

There are several XML standards such as Document Type Definitions (DTD), Schemas, Style sheets, XML Documents etc. For OSP testing purposes style sheets are not relevant and XML Schemas are not used. In the present document we shall therefore concentrate on: DTD and XML documents themselves.

The declaration of TTCN3 types is explained by the example of the XML AuthorizationRequest component.

6.3.1 AuthorizationRequest

6.3.1.1 XML declaration

The structure of the XML AuthorizationRequest component has the following characteristics:

a) there are two basic parts, the AttributeList part and the Element part.

b) within the Element part the elements appear in a given order.

c) the elements may appear multiple times.

```
<!ELEMENT AuthorizationRequest ( Timestamp, CallId+, SourceInfo, SourceAlternate*,
    DestinationInfo, DestinationAlternate*, Service, MaximumDestinations, Token*,
    SubscriberAuthenticationInfo* )>
<!ATTLIST AuthorizationRequest componentId ID #REQUIRED>
```

6.3.1.2 TTCN3 type

The TTCN-3 type represents the structure of the XML AuthorizationRequest component:

a) the two basic parts are represented in **type record AuthorizationRequest**. For testing purposes the payload is added.

b) all elements are represented in **type record AuthorizationRequestContents**.

c) the element CallId may appear multiple times. Its type is: **type record of length(1 .. 255) CallId CallId_List**.

```
type record AuthorizationRequest
{
AuthorizationRequestAttribute       authorizationRequestAttribute,
AuthorizationRequestContents        authorizationRequestContents,
AuthorizationRequestPayload         authorizationRequestPayload
}

type record AuthorizationRequestAttribute
{
Int255      messageId,
PDUType     pduType (AUTHORIZATIONREQUEST_E),
Int255      componentId
}

type record AuthorizationRequestContents
{
Int1                                numberTimestamp (1),
Timestamp                           timestamp,
Int255                              numberCallId,
CallId_List                         callId,
Int1                                numberSourceInfo (1),
SourceInfo                          sourceInfo,
Int255                              numberSourceAlternate,
SourceAlternate_List                sourceAlternate optional,
Int1                                numberDestinationInfo (1),
DestinationInfo                     destinationInfo,
Int255                              numberDestinationAlternate,
DestinationAlternate_List           destinationAlternate optional,
Int1                                numberService (1),
Service                             service,
Int1                                numberMaximumDestinations (1),
MaximumDestination                  maximumDestinations,
Int255                              numberToken,
Token_List                          token optional,
Int255                              numberSubscriberAuthenticationInfo,
SubscriberAuthenticationInfo_List   subscriberAuthenticationInfo optional
}

type record AuthorizationRequestPayload
{
Int255      payloadlength,
Char255     payload
}
```

6.3.1.3 TTCN3 template

A TTCN-3 template for receiving a XML AuthorizationRequest component is shown:

a) each field of **authorizationRequestAttribute**, **authorizationRequestContents** and **authorizationRequestPayload** has a value or a wildcard assigned;

b) elements are received either with wildcards or a certain value;

c) the element **CallId** shall be received one time.

```
template AuthorizationRequest AutReq_R1:=
{
    authorizationRequestAttribute:=
    {
        messageId:= ?,
        pduType:=   AUTHORIZATIONREQUEST_E,
        componentId:=   ?
    },
    authorizationRequestContents:=
    {
        numberTimestamp:= 1,
        timestamp:= Timestamp_R1,
        numberCallId:= 1,
        callId:= CallId_R1,
        numberSourceInfo:= 1,
        sourceInfo:= SourceInfo_R1,
        numberSourceAlternate:= ?,
        sourceAlternate:= *,
        numberDestinationInfo:= 1,
        destinationInfo:= DestinationInfo_R1,
        numberDestinationAlternate:= ?,
        destinationAlternate:= *,
        numberService:= 1,
        service:= Service_R1,
        numberMaximumDestinations:= 1,
        maximumDestinations:= MaximumDestinations_R1,
        numberToken:= ?,
        token:= *,
        numberSubscriberAuthenticationInfo:= ?,
        subscriberAuthenticationInfo:= *
    },
    authorizationRequestPayload:=
    {
        payloadlength:=?,
        payload:= ?
    }
}
```

6.3.2 Parameterization

TTCN-3 templates can be parameterized. For example, the messageId, the componentId, the currency in use, the amount (number of currencies being accounted), the increment (number of units being accounted) and the units (by which pricing is measured) are parameterized in the PricingIndication PDU.

```
template    PricingIndication   PriInd_S1(integer msgId,integer compId,Char3 p_currency,
        float p_amount, integer p_increment,UnitContents p_unitContents):=
        {
            pricingIndicationAttribute:=
            {
                messageId:= msgId,
                pduType:= PRICINGINDICATION_E,
                componentId:= compId
            },
            pricingIndicationContents:=
            {
                numberTimestamp:= 1,
                timestamp:= Timestamp_S1,
                numberSourceInfo:= 1,
                sourceInfo:= SourceInfo_S1,
                numberDestinationInfo:= 1,
                destinationInfo:= DestinationInfo_S1,
                numberCurrency:= 1,
                currency:= Currency_S1(p_currency),
                numberAmount:= 1,
                amount:= Amount_S1(p_amount),
                numberIncrement:= 1,
                increment:= Increment_S1(p_increment),
                numberUnit:= 1,
                unit:= Unit_S1(p_unitContents),
                numberService:= 1,
                service:= Service_S1,
                numberValidAfter:= 1,
                validAfter:= ValidAfter_S1,
                numberValidUntil:= 1,
                validUntil:= ValidUntil_S1
            },
            pricingIndicationPayload:=
            {
                payloadlength:=0,
                payload:= omit
            }
        }
```

6.3.3 Wildcards

The TTCN-3 matching mechanisms (wildcards) can also be used.

For example, when receiving a Status element, then:

- any criticalAttribute,
- any code shall,
- any numberDescription,
- any or no description

shall be received.

```
template Status Status_R1:=
        {
            criticalAttribute:= ?,
            statusContents:=
            {
                code:= ?,
                numberDescription:= ?,
                description:= *
            }
        }
```

7 Practical experience of using TTCN-3

In the context of ETSI the practical use of TTCN-3 has been demonstrated on at least two occasions:

- The IPV6 Interoperability Events (October 2000 and November 2001) where Ericsson demonstrated their (prototype) TTCN-3 based test platform by executing a hundred or so IPV6 TTCN-3 tests against a number of different IPV6 Routers.
- The 7th SIP Bake-off (March 2001) where TestingTech demonstrated the execution of a small number (but fully-functional) TTCN-3 SIP tests against a real SIP implementation. At the same event, ACACIA also performed conformance testing of a number of SIP implementations. These tests were actually written in TTCN-2, which shows that the techniques work even if the more elegant solution is TTCN-3.
- At IMTC/TTC Winterop in Kobe Japan a conformance testing facility was organized by TestingTech using approximately 100 TIPHON SIP tests in TTCN-3.
- For examples of how the ideas in the present document have been applied see the ETSI test suites for SIP and OSP.

8 Availability of tools

Two types of tools are needed:

- Development Environments (DEs) with editors, syntax checkers etc. for writing and checking the TTCN-3 test suites, and
- TTCN-3 compilers and real SIP test systems on which the test suites will be executed.

The Specialist Task Forces at ETSI will require the first type of tool. At the time of writing the present document there exists at least one editor combined with a simple syntax checker and built-in support for Test Purposes (TTCN-3 StarterPac). A command-line TTCN-3 syntax checker (part of the Debian GNU/Linus package) is also available from the University of Luebeck (Belgium). This tool, which runs under Windows as well as Linux, can be linked to TTCN-3 customized versions of editors such as EMACS and Edit++. ETSI already has access to all these tools so support in that area is adequate, at least for the time-being. Commercial editors and sophisticated development environments are expected Q3 - Q4 of 2001 from companies such as Telelogic, DaVinci Communications and TestingTech.

NOTE: There will also be support for the Tabular and Graphical Presentation Formats of TTCN-3

It is expected that the ETSI Secretariat (via the PTCC) will keep track of developments in that area so that the STFs have the best possible support available.

Generation of executable test suites (not done by ETSI) will require TTCN-3 compilers. SIP test systems will also need to implement the low-level processing (encoding/decoding) describe in clause 5 of the present document. One commercial TTCN-3 compiler (from TestingTech) is already available and others are expected later in 2001.

Supporters of this work such as Nokia and Ericsson have (or are developing) their own in-house TTCN-3 tools. It is not yet known whether these tools will be made available on a commercial basis.

Finally, it is very important that producers of the TTCN-3 test specifications liase closely with these, and indeed all, tool makers to ensure that the test suites are implementable in a reasonable manner on the real equipment that is currently available.

9 Maintenance of the TTCN-3 standard

TTCN-3 [1] is an ETSI Standard which will also be published as ITU-T Recommendation Z.140 [4]. ETSI Technical Committee MTS (Methods for Testing and Specification) is committed to the development and maintenance of TTCN-3. In particular, an STF (Specialist Task Force) is in place to support maintenance of the standard in 2001 and the early part of 2002. It is planned that at least some level of MTS/STF support for maintenance will continue in the future on an as needed basis.

10 Training

It is expected that the PTCC (ETSI's Protocol and Testing Competence Centre) will provide the necessary training of the STF experts involved in the production of the TIPHON TTCN-3 test specifications. Assuming a good working knowledge of TTCN-2 existing users should pick-up TTCN-3 without much difficulty.

It is understood that MTS is considering the production of TTCN-3 educational material. This initiative should be encouraged.

Annex A: Suggested style guidelines

A.1 Introduction

This rule and guideline are given to promote good style in the development of TTCN-3 Abstract Test Suites (ATS). It is by no means mandatory but is to be seen as a tool available to a Special Task Force leader responsible for a TTCN-3 project. This leader will be faced from the start with style decisions, especially if the team is composed of a mix of members from the C-type languages community and the TTCN-2 community. Hopefully, this guide will help the leader in the first steps along this thorny decision path. If nothing else, it highlights the style issues that will require resolution.

Whether in TTCN-3 or any other language, good programming style is consistent, easy to read and understand, and free of common style errors recognized by any language community.

Good style is an aid to both the ATS writer and reader. The writer's creativity is not lessened by good style. In fact, good style coupled with a creative solution often yields elegant code.

The reader benefits from style by knowing the means that will show key items within the code. Uniform style lessens confusion on how to interpret code. Style has special importance because it improves readability. Readability is especially important in the testing community since the human tester must often understand the test case code in order to understand why a test case failed. Thus, in the testing environment, there is an additional class of code readers other than the maintainer; i.e. the tester. The ultimate success of TTCN-3 code may very well depend upon its tester readability rather than writer or maintainer readability.

There are many guidelines but only one rule and its two corollaries. And, this rule is not really mandatory since this annex is only informative!

The guidelines are recommendations to the writer. In some cases, several options given. Hopefully, the writer will chose one of them. In other cases, there is only one example shown. If the writer does not like it, feel free to "roll your own." Of course, the One Rule applies to this new style.

A.2 The rule and its two corollaries

The Rule
The same coding style shall be used throughout the ATS.

For example, only one commenting style shall be used throughout the same document. Mixed styles are unacceptable.

First Corollary
Maintenance of an ATS will adhere to the coding style used in the original ATS. (Mixed coding style is harder to maintain than poor style.)

Second Corollary
Every time The Rule is broken, it must be clearly justified in clear documentation presented at the beginning of the ATS.

A.3 Some guidelines

A.3.1 Module organization

The ATS will be organized into a hierarchical structure. First, by use of the intrinsic TTCN-3 separation between the Definitions and Control Parts of the module. And, secondly, by use of the `group` reserved word. The latter structuring mechanism is not strictly formal in this version of TTCN-3 since the scope of the items in the newly defined groups may extend to other groups outside of the original. One must understand grouping as a conceptual structure within the ATS writer's mind rather than the organization being in the code itself.

A proposed module organization follows. It amplifies the strict TTCN-2 organization by incorporating some TTCN-3 features. Different group names and structures are possible. In any event, this Test Suite Structure (TSS) must be specified in detail in an ETSI document. This will give the writer the opportunity to explain and justify in detail the test suite structure as shown in the TTCN-3 grouping.

In the following organization, indentation indicates sublevels in the structure. Items with the same indentation are equal within the hierarchy.

```
ImportedDefinitions
SystemConfiguration
  TestComponents
    MTC
    PTCs
  Ports
  ProtocolMessages
Types
  SimpleTypes
    SimpleConstants
    SimpleVariables
  CompositeTypes
    CompositeConstants
    CompositeVariables
  MessageComponentTypes
  MessageTypes
    AbstractServicePrimitives
    CoordinationMessages
    PDUs
    Messages
    Datagrams
  MultiMessageTypes
Templates
  SimpleTemplates
  CompositeTemplates
  MessageComponentTemplates
  MessageTemplates
    ASPtemplates
    CMtemplates
    PDUtemplates
    MessageTemplates
    DatagramTemplates
  MultiMessageTemplates
Functions
  TestCaseSelectionExpressions
  TestGroupSelectionExpressions
  ATSoperations
Behaviour
  BehaviourFunctions
  Preambles
  Postambles
  TestSteps
  Defaults
  TestCases
    Side1
      Side1ValidBehavior
Side1InvalidBehavior
Side1InopportuneBehavior
Side1IncorrectSyntaxHandling
Side1ErrorHandling
    Side2
      Side2ValidBehavior
      Side2InvalidBehavior
      Side2InopportuneBehavior
      Side2IncorrectSyntaxHandling
      Side2ErrorHandling
```

Some explanatory comments follow:

`ImportedDefinitions`
This clause contains all TTCN-3 code which uses the reserved word `imported` in the definitions. This permits the reader to see in one place all references to definitions which are outside of the current ATS.

```
   Example:  import all from AnASN1Module language "ASN.1:1997";
```

`SystemConfiguration`
This group attempts to unify all definitions concerning the configuration of the System Under Test into one hierarchy. This group appears intuitively to be valid across all systems under test and, thus, should be consistent across all ATSs.

`ProtocolMessages`
This subgroup simply defines the message identifiers, their types, and their `in` and `out` qualifiers. It does not give the actual list of the elements composing the messages. This is done later in the `MessageTypes` group.

`SimpleTypes`
This subgroup contains only those definitions and constants which use only one of the basic types of TTCN-3. Structured types or non-basic TTCN-3 types are not used herein. Enumerations and unions are not considered basic types in this structure.

`CompositeTypes`
This subgroup contains structured, union, and enumerated types which are not actual messages. It does include message elements such as information elements familiar to the ISDN world or attributes familiar to the IETF type protocols. The definition of messages is in the `MessageTypes` group.

`MessageComponentTypes`
The word "message" means different things in different protocols. Its context here is the largest data structure sent between two communicating entities for a given protocol level of the entities. A Message Component is an element of a message at one level below the message. For ISDN-like protocols, it could be an Information Element. For IP-like protocols, it could be an Attribute. Messages are usually record-like structures of different elements. One of these elements is a message component. Of course, as is the case with the other guidelines, the ATS writer may eliminate this level of abstraction.

`MessageTypes`
This is a very sensitive subject. The word "message" means different things in different protocols. Its context here is the largest data structure sent between two communicating entities for a given protocol level of the entities. In ISDN-like protocols, these messages are named PDUs. In IP-like protocols, these messages are called datagrams. In other messages, these messages are called "messages". One can see where there might be some confusion. The subgroups shown in the example hierarchy do not all have to exist. Some protocols will use one of the subgroups while others will use another completely different subgroup. The list of `MessageTypes` subgroups is not exhaustive. The appropriate subgroup name will often be found in the protocol's specification.

`AbstractServicePrimitives`
Abstract Service Primitives are a pure TTCN-2 concept. If the writer does not like it, she does not have to use it. However, if the writer chooses to wrap the messages in a function with other parameters and then send and receive this function, the chances are that the function is an Abstract Service Primitive in the TTCN-2 sense.

`CoordinationMessages`
Coordination Messages is a valid semantic concept in both TTCN languages. They are used in modelling and testing concurrent activity or two or more equivalent items under test.

`PDUs, Messages,` and `Datagrams`
See the above discussion on `MessageTypes.`

`MultiMessageTypes`
One sees, at times, messages composed of messages, e.g. in IETF-like protocols. For example, a message sent from one peer to another may contain messages from the same protocol. This group attempts to abstract these messages into a separate semantic group.

`Templates`
The substructure of the templates group should mirror the structure of the `MessageTypes` group. In an addition, editing tools should allow the reader/writer to shift back and forth between a template and its corresponding type definition. This is especially useful for the template writer who can use the type definition as a basic structure for building the template.

`Functions`
All functions should be placed into a hierarchy and located in one place in the ATS.

`TestCaseSelectionExpressions`
Test case selection expressions are a pure TTCN-2 construct and concept which has been grafted onto this structure. The running or not of a certain test case is often based on configuration information or other parameters shown in the PICS document. The boolean conditions for running or not running a test case should be grouped into a test case selection expression group.

`TestGroupSelectionExpressions`
The same type of logic exists for this group as that for `TestCaseSelectionExpressions`. Groups of test cases are often not run because of configuration options not being implemented and documented in the PICS.

`ATSoperations`
The ATS writer will be writing many functions in TTCN-3. This is one of the language's greatest opportunities. These functions should be grouped into logical units if at all possible. Possible subgroups are not shown in the proposed hierarchy but should be developed by the writer. If no hierarchical structure is possible or advisable, present the functions in alphabetical order under this group.

`Behaviour`
The subgroups under `Behaviour` are largely taken from TTCN-2 and could apply to any TTCN-3 code. `Preambles, Postambles, TestSteps, Defaults,` and `TestCases` are all TTCN-2 constructs and semantics but would seem universal in their application as TTCN-3 functions. These are, in effect, logical test groupings of TTCN-3 functions. If the writer has another hierarchy of functions which are related to behaviour, it could be given in the `BehaviourFunctions` group. A C-like programmer could under `TestSteps` as subroutines called within the test case.

`BehaviourFunctions`
See the `Behaviour` group comments.

`TestCases`
An ATS usually contains the test cases for both peers of the protocol. The Test Suite Structure document defines this structure. This structure should be exactly mirrored in the ATS itself.
Since TTCN-3 groups do not define scoping limits but are only a structuring tool, the names of each group must be different from any other name or a compiler error will result. For example, one cannot use the subgroup name `ValidBehavior` under the group `Side1` and the same subgroup name under the group `Side2`. The names themselves must be different; e.g. `Side1ValidBehavior` and `Side2ValidBehavior`. This scoping limitation may be changed in a future TTCN-3 version.

A.3.2 Comments

There are several recommended alternatives for block comment style shown below. Remember to use The Rule and employ the same block comment form throughout an ATS. Do not change the comment style within an ATS. Block comments are tabbed over to the same column as the item of code they describe.

```
  /*
   *  Here is an example block comment.
   *  The comment text should be tabbed or spaced over uniformly
   *  to the same column as the code it describes.
   *  The opening slash-star and closing star-slash are alone on a line.
   *  This comment explains the code immediately below.
   */
  function
  getReplyAddr (in Via locVia, address locAddress)
  runs on SipComponent
  return address {
    /*
     *  This comment substitutes for the operations
     *  defining the function.
     */
    return locAddress
}

  //
  //  Another example format for block comments.
  //  The comment text should be tabbed or spaced over uniformly
  //  to the same column as the code they describe.
  //  The opening slash-slash and closing slash-slash are alone on a line.
  //  This comment explains the code immediately below.
  //
  function
  getReplyAddr (in Via locVia, address locAddress)
  runs on SipComponent
  return address {
    //
    //  This comment substitutes for the operations
    //  defining the function.
    //
    return locAddress
}
```

There are also several alternatives for in-line comments. Again, use the same format throughout an ATS. Do not change formats within an ATS. Just like block comments, these should also be tabbed over to the same column as the item they describe.

One alternatives for an in-line comment:

```
  getReplyAddr (in Via locVia, address locAddress)
  runs on SipComponent
  return address{
    /*  This is in-line comment is for the line below. */
    locaddress := addressStub;
    return locAddress
}
```

Another alternative for an in-line comment:

```
  getReplyAddr (in Via locVia, address locAddress)
  runs on SipComponent
  return address{
    //  This is an in-line comment is for the line below.
    locaddress := addressStub;
    return locAddress
}
```

End-of-line comments can also appear in one of several forms. The only requirement is to select one of them, or make your own, and then follow The Rule. If more than one such short comment appears in a block of code, they should align on the same column.

One way of writing in-line comments:

```
  if (myVar < 10) {
     myVar := myVar + 10;                     /*  Must align on the below column.  */
     log("myVar was < 10 but not any more")   /*  log does not mean logarithm here */
  }
  else {
     myVar := myVar/5                         /*  Still in the same code block.    */
  }

And another away:
  if (myVar < 10) {
     myVar := myVar + 10;                     //  Must align on the below column.
     log("myVar was < 10 but not any more")   //  log does not mean logarithm here
  }
  else {
     myVar := myVar/5                         //  Still in the same code block.
  }
```

Comments can be thought of as either strategic or tactical. A strategic comment describes what a function or a set of code is intended to do and is placed before this code. A tactical comment describes what a single line of code is intended to do and is placed, if possible, at the end of this line. Unfortunately, too many tactical comments can cause clutter and, at times, make code unreadable. For this reason, it is recommended to primarily use strategic comments unless trying to explain very complicated code.

Use comments to justify offensive code; i.e. hacks. The comment should explain the behaviour requiring the hack and why the hack is a good fix. In testing, hacks may be required for test equipment performance reasons.

A.3.3 Type definitions

A.3.3.1 Basic types

Unrelated definitions, even of the same type, should be on separate lines. If types are grouped and have a similar structure, the elements should align on the same column. Similar basic type definitions can be informally grouped under an explanatory comment. Dissimilar basic type definitions should be separated by an empty line.

```
/* Basic type definitions.  Note the column alignment. */
type charstring Char1   length(1);
type charstring Char2   length(2);
type charstring Char20  length(0 .. 20);
type charstring Char255 length(0 .. 255);
type charstring Char511 length(0 .. 511);

type octetstring Oct16 length(16);

type integer Int1   (0 .. 1);
type integer Int20  (0 .. 20);
type integer Int255 (0 .. 255);
type integer Int511 (0 .. 511);

/* List type definitions*/
type record of length(1 .. 255) AuthorityURL   AuthorityURL_List;
type record of length(1 .. 255) CallId        CallId_List;

type bitstring AbitString       length(9);
type bitstring AnotherBitString length(2);

type charstring AcharString       length(2);
type charstring AnotherCharString length(300);

type AthirdCharString AnotherCharString;
```

A.3.3.2 Structured types

A.3.3.2.1 Enumerations

The type name is placed at the left margin to improve its "findability". The specific numerations are indented and align on the same column. Explicit enumeration values immediately follow the enumerated field name. Each enumeration shall have its own separate line.

Although not specifically a style issue, all enumerated values must have unique field names within the same scope level. For example, one cannot use the enumeration field name `pricingConfirmation` in another enumeration definition at the same scoping level. This may be changed in later versions of TTCN-3.

```
type enumerated
OSPmessageType{
    pricingIndication(33),
    pricingConfirmation,
    authorizationRequest,
    authorizationResponse,
    authorizationIndication,
    authorizationConfirmation,
    usageIndication,
    usageConfirmation
}
```

A.3.3.2.2 Records et al

The type name is placed at the left margin to improve its "findability". Each record's element occupies a separate line, is indented from the record name column and aligned on the same column. Each field name is also column-aligned. Value initializations are also separated from the previous item by one space.

It is sometimes easy to miss the `optional` reserved word if they are sparse within the record and close to the right margin. Thus, `optional` is separated from the field name by one space. It is not column aligned.

```
type record
PricingConfirmationAttribute{
  Int255       messageId,
  MessageType  messagetype (PRICINGCONFIRMATION),
  Int255           componentId optional
}
```

Unions are similar to records. Note that field initialization cannot occur within union definitions

```
type union
MyUnionType {
  integer    number,
  charstring string
}
```

A.3.3.2.3 Variable definitions

Arrays

Arrays are seen as variables in TTCN-3 and not as types.

```
  //  Given
  type record MyRecordType{
    integer       field1,
    MyOtherStruct field2,
    charstring    field3
  };

  //  An array of MyRecordType could be
  var MyRecordType MyRecordArray[10];
```

Initialization

Default initialization values are not required for variables in the TTCN-3 specification. Uninitialized variables at run time shall cause a test case error. Variables are defined only in module control, test cases, and functions. Otherwise said, variables which are not assigned values by expressions or other operations must be explicitly initialized.

Every variable that is defined is given a value before used. If possible, always use initialization instead of assignment.

```
var integer myVariable := AsymbolicValue;
```

Only one intialization should occur on one line. Multiple initializations on the same line are more difficult to scan. For example, write

```
var boolean myFirstBooleanVar  := true;
var boolean mySecondBooleanVar := true;
```

rather than

```
var boolean myFirstBooleanVar := true, mySecondBooleanVar := true;
```

Avoid the uses of numeric values except in obvious cases like iterations or loop counters. Represent them with symbolic values using `const` or `var`.

A.3.4 Function definitions

Functions should be grouped into some sort of meaningful order. Top-down is generally better than bottom-up, and a "breadth-first" approach (functions on a similar level of abstraction together) is preferred over depth-first (functions defined as soon as possible after their calls). For example, one may create a group of test suite operations which may require other functions to be called within them. All the test suite operations should be grouped and the sub-functions called within them should be separated from and after the calling functions. Judgement is especially required here.

If defining large numbers of essentially independent utility functions, consider alphabetical order.

Each function should be preceded by a block comment prologue that gives a short description of what the function is and (if not clear) how to use it. Avoid duplicating information clear from the code.

If the function returns a value, the `return` Type part of the function header should start in the same column as the function name. This permits easy identification of the type of the value the function returns. If the value returned requires a long explanation, it should be given in the prologue; otherwise it can be tabbed over on the same line as the return type.

If a function does not return a value, then the return type is implicitly void. The reserved word `void` does not exist in TTCN-3. The absence of a `return` reserved word in a function header indicates that the function returns no value. There is no default return type for a function.

The function name and the formal parameter list should be alone on a line and begin in column 1. If the parameter list is too long for one line, it shall be broken into successive lines with the first character of each succeeding line aligned on the column of the first character of the first parameter.

Avoid long and complex functions. By definition, if the function is difficult to understand, it is too long. Shorter functions allow errors to identified more quickly and easily.

Internal variable declarations are grouped at the start of the function body. They are not interspersed throughout the function's execution part code. The variables are separated from the function's execution part by a clear line.

Comments should start in the same column and over the line(s) of code for which they apply.

```
//
//  It is unclear from the code snip what kind of address is being
//  returned.  This header should give this kind of information.
//  getReplyAddr returns a value of type address as shown in the
//  header line "return address"
//
function
getReplyAddr (in Via     locVia,
              Address    locAddress,
              Parameter1 param1,
              Parameter2 param2,
              Parameter3 param3,
  Etcetera   etc )
runs on SipComponent
return Address{
   var integer    aNumber;
   var charstring aCharString;
   var Address    tempAddress;
```

```
      //  The return address shall be calculated in this
      //  execution section.
      //  This is the code
      return locAddress;
   }
```

A.3.5 Whitespace

Whitespace has much more semantic meaning to a human TTCN-3 reader than to a compiler. In this context, whitespace may have even more meaning than commas, semicolons, et al.

There should be at least 2 blank lines between the end of one function and the comments for the next.

A long string of conditional operators should be split into several lines. Similarly for elaborate loop condition expressions.

```
if (anAddress == null
   and total < needed
   and needed < maxAllotment
   and serverActive(currentInput)  ) {
           findAnotherServer(serverAddress)
} else {
   useThisServer(thisServerAddress)
};
```

Tabulation

Indenting is done by tabulation, not by spacing over. Tabulations may include either 2, 4, 6, or 8 spaces. Some exceptions may occur under certain structures like in `if` statements where three spaces makes more sense. See above. Whatever tabulation rules are adopted shall be applied across the entire ATS.

A.3.6 Statements

A.3.6.1 Simple statements

There should be only one statement per line.

The null body of a `for` or `while` loop should be alone on a line and commented so that it is clear that the null body is intentional and not missing code.

```
for ( j := 1; booleanDatabaseElement[j]; j <= 10 )
     {  // VOID, just want to find the index of the first true element };
```

A.3.6.2 Compound statements

A compound statement is a list of statements enclosed by braces.

There are many common ways of formatting the braces. Each style has lead to religious wars. A recommended style (closely following Kernigan & Ritchie) is shown below for different situations. In any case, the church and the state should be separated to reduce the possibility of future crusades, holy wars, whatever. Remember there is only One Rule: apply the braces style uniformly across the same ATS.

```
//  an if statement without a following else
if (condition) {
   statement;
   statement
};

//  an if-else statement
if ( condition ) {
   statement;
   statement
} else {
   statement;
   statement
};

for ( initial; condition; nextConditionValue ) {
  statement;
  statement
};

while ( condition ) {
  statement;
  statement
};

do {
   statement;
   statement
}  while ( condition );
```

An `if-else` with `else if` should be written with the `else` conditions left-justified. The format then looks like a case or switch construct from other languages.

```
if ( firstCondition ) {
   statement;
   secondStatement
} else if ( secondCondition ) {
   thirdStatement;
   fourthStatement
} else if ( thirdCondition ) {
  fifthStatement;
  sixthStatement
} else {
  defaultStatements;
};
```

A.3.7 Naming conventions

Constants should be in all `CAPS`.

The following begin in lower case (unless the identifier used is a proper noun):

- enumerated values;
- function names;
- record field names;
- union field names;
- variable names.

Defined types are `Capitalized`. Field names within a defined type begin in `lowerCase`. The type names within a structured type are `Capitalized`. This allows some consistency with TTCN-2 practice and ASN.1 requirements. There would be much confusion if an ASN.1 module is used which requires types `Capitalized` and TTCN-3 definitions module which has types in `lowerCase.` The reader could not know immediately if he was looking at a type or field name in the code. The mixed conventions also violate The Rule.

Even if TTCN-3 is case-sensitive, avoid names that differ only in case, like `foo` and `Foo`. Similarly, avoid using two names differentiated only by an underscore like `foobar` and `foo_bar`. The potential for confusion is considerable.

Similarly, avoid names that look like each other. In many fonts, "`l`", "`1`", and "`I`" look quite similar (don't they?). A variable named "`l`" is particularly bad because it looks so much like "`1`". The same applies for "`0`" and "`O`".

Identifiers must start with an alphabetical character. They can include underscores "_" after the first character. Identifiers are often composed of several words joined together into one character string. It is recommended to use the style

```
aCompoundName
```

rather than

```
a_compound_name.
```

Sometimes it is necessary to vary the presentation to increase readability. For example:

```
thisIsAcompoundName
```

and

```
thisIsApduMessage
```

are more readable than

```
thisIsACompoundName
```

and

```
thisIsAPDUMSG.
```

Avoid abbreviations in identifiers unless they are known by all in the art. PDU may be obvious to OSI types but not so in the Internet protocol world. And the two worlds meet in ATSs! The same type of confusion results in the abbreviation MSG.

Other examples of name choices:

```
var integer groupID;     // instead of grpID
var integer nameLength;  // instead of namLn
```

The following variable name is ambiguous:

```
var Address termProcess;  //  Terminate process or terminal process?
```

When dot notation is used in an identifier, do not place spaces between the names and the dot. For example, this:

```
  a.good.dot.notation
```

but not this:

```
  not . so . good . dot . notation
```

One may encounter long identifier in dot notation which may require continuation onto the next line. One style is to continue the identifier in the column below the first dot on the following line. For example:

```
one.style.is.to.continue.the.identifier.in.the.column.below.the
   .first.dot.on.the.following.line
```

A.3.8 Beautifiers and formatters

EMACS has a useful TTCN-3 formatter which, no doubt, has rules different than these. But do not rely on any automatic beautifier and formatter. One person who benefits from good program style is the ATS writer, especially in the early design of handwritten algorithms or pseudocode. The need for attention to white space and indentation are greatest during the test development stage. Test developers can do a better job of making clear the complete visual layout of a function or file with the normal attention to detail of a careful developer. Beautifiers cannot read our minds.

A.3.9 Presentation fonts and sheet orientation

TTCN-3 should be written in a non-proportional font such as Courier. How else to get easy column alignment? Proportional fonts should not be used.

Font size is at the discretion of the developer. However, the size should be large enough so that the code is easily readable when printed.

Printing may be either in "portrait" or "landscape" sheet orientation. Some of the long lines seen in test cases may lead one to think about landscape mode for test cases and portrait mode for definitions. This would violate The Rule.

A.3.10 Alternates, named or not

Alternative behaviour is one of the most important TTCN-3 capabilities and yet one of the most difficult items to make readable. Nested alternative behaviour is particularly difficult to convincingly portray. Considerable effort is required for their formatting. First experience has shown that the brackets "[]" are a key element to the eye for identifying alternate behaviour. Their use coupled with indentation, braces, and other whitespace can provide the meaning to the eye that the reader and writer need.

The placement of matching braces is particularly difficult when alternatives become nested; more so when the alternatives occur two or three levels deep in a test case.

A readable style for nested alternates is suggested in the following example. The comments are for explanatory purposes only and could be omitted in real code. But, they are useful to determine where one `alt` construct leaves off and the other kicks in.

```
alt {                                    // alt level 1
[] sipU.receive  (iNVITErequest_r_1) -> value iNVITErequest {
     recCallId := iNVITErequest.reqHeader.callId;
     recCSeq := iNVITErequest.reqHeader.cSeq;
     recFrom := iNVITErequest.reqHeader.fromField;
     recTo := iNVITErequest.reqHeader.toField;
     recVia := iNVITErequest.reqHeader.via;
     iutAddr := getResponseAddr (recVia);
     sipU.send (response_200_s_1(recCallId, recCSeq,
                            recFrom, recTo, recVia )) to iutAddr;
     TAck.start(TIMER_ACK);
     alt{                                // alt level 2
     [] sipU.receive ( aCKrequest_r_1 (
                        recCallid,
                           recCSeq, ?, recTo, ? )) -> value aCKrequest {
          verdict.set(pass);
          terminateCall ();
        }
     [] sipU.receive {
          all timer.stop;
          verdict.set(inconc);
          cancelCall();
          }
     [] Twait.timeout {
          verdict.set(inconc);
          cancelCall()
        }
     }                                   // endalt level 2
   }                                     // end sipU.receive at alt level 1
[] sipU.receive {
     all timer.stop;
     verdict.set(inconc);
     stop
   }
[] sipM.receive {
     all timer.stop;
     verdict.set(inconc);
     stop
   }
[] Twait.timeout {
     verdict.set(inconc);
     stop
   }
 } // endalt level 1
```

If there is no match in a named alternative or in an `if`, it is excellent style to place actions to take at the end of the `alt/if` construct.

A.3.11 Calls and references to other modules

Any calls or references to external modules should be placed at the beginning of the ATS immediately following the ATS module's header. For example:

```
  module Style ( )
  {
  /* External calls and references begin here */
  import all from AnASN1Module language "ASN.1:1997";


  /*
   *  This is the definitions part.
   */

  control
     {
     }
  }
```

A.3.12 Test case style

Test cases are usually where all the action is and where the most complex and lengthy code has a natural tendency to accumulate. Thus, a special effort here at abstraction and layout will pay off for both the test writer and her readers.

An example follows. Some explanatory comments are given afterwards.

```
/*
 *  The test purpose appears here as commented text.
 */
testcase SS_OE_CE_006 ()
runs on ProtocolComponent
system ProtocolInterfaces {

  var INVITErequest  iNVITErequest;
  var INVITEresponse iNVITEresponse;
  var ACKrequest     aCKrequest;
  var IUTaddr        iutAddr;

/*  PREAMBLE  */
uaRegister();

sut.action ( "Please send INVITE" );
TWait.start(TIMER_WAIT);
alt {
[] pcoU.receive ( iNVITErequest ) {
     mapToInviteResponse ( iNVITErequest, iNVITEresponse );
     mapToAckRequest (iNVITEresponse, aCKrequest );
     iutAddr := getResponseAddr( iNVITErequest );
     pcoU.send ( iNVITEresponse ) to iutAddr;
     TAck.start( TIMER_ACK );
     alt {
     [] sipU.receive ( aCKrequest ) {
          verdict.set(pass);
          terminateCall ();
        }
     [] sipU.receive {
          all timer.stop;
          verdict.set(inconc);
          cancelCall();
        }
     [] Twait.timeout {
        verdict.set(inconc);
        cancelCall()
        }
     /*  nested alt ends here */
     }
   /*  pcoU.receive in first alt ends here */
   }
[] sipU.receive {
     all timer.stop;
     verdict.set(inconc);
     stop
     }
[] sipM.receive {
     all timer.stop;
     verdict.set(inconc);
     stop
     }
[] Twait.timeout {
     verdict.set(inconc);
     stop
     }
}  /* first level of alt */
}  /* end of test case */
```

The Test Purpose is copied verbatim from the Test Purpose specification and placed as the first element in any test case. The test case number is usually found in the same document. Test case numbering is not addressed in this guide.

The test case line starts in column 1 of the page. Horizontal space becomes critical in test cases because of the length produced by name dot notation, long parameter lists, and indentation.

If a parameter list is long, it should be broken into separate lines lining up on the same column as the first character in the first parameter. Refer to the example in alternates, named or not section of this guide for a suggested manner to portray long parameter lists. For example:

```
  pco.send ( aTemplate( firstParameter, parameter2,
                        parameter3, parameter4,
                        parameter5, parameter6,
                        parameter7, parameter8,
                        parameter9 )) {
    firstStatement;
    statement2
  );
```

If all these parameters are necessary for the test case's control, then they should be visible. However, such a case is not likely. It is highly recommended to make parameters visible only when they have something to do with test case control and decision-making. If they are not used for test case control or are not related to the test purpose, it is highly recommended to abstract them out of the test case by using a type which subsumes them.

The `runs on` line is mandatory and starts in column 1.

The `system` line is also mandatory and starts in column 1.

The opening brace is placed according to the specific brace style chosen for the entire ATS. Refer to the religious wars section.

A blank line separates the test case header and the definitions. All definitions used within the test case are placed immediately after the header. None are placed within the test case behaviour part. These items are indented on a tab stop.

Another blank line separates the definitions and behaviour parts. The behaviour part begins alignment on the 1st column to signal the start of the behaviour section and to reduce the page width problem. Note that portrait or landscape orientation may be used. Usage must be the same throughout the ATS.

Use a comment to indicate the preamble part if that is not obvious in the function name serving as the preamble. A blank line should separate the preamble(s) from the actual test case code.

Timers are sequential items in TTCN-3 rather than being associated with a message as in TTCN-2. They may be declared anywhere within in the sequence as the writer deems appropriate.

Only one statement per line should be written in a test case. The situation is complicated enough as it is.

See the clause concerning alternates for style guidance.

In TTCN-2 style, one often sees numerous assignment expressions immediately after a receive in order to map the incoming data into send data. These leads to long test code which provides little key information during testing. It may be better to abstract these assignments into a function which maps incoming data into transmit data. If data is needed for test case decisions or for significant parameters, it can be derived independently using a separate function.

One must be very careful of statements which occur, if any, after any `alt` construct. It would be much better to avoid them like the plague. It is very good practice to ensure that the test case will ultimately stop in each alt possibility. If alternates are nested or if functions are called from the alternates, a stop construct should eventually be at the end of the tree. One can used a named alternate as a default to stop test case execution as well. The named alternates which are active must be highly visible within the test case to know which alternate applies. A bracketed else `[ else ]` may also be used as the last possibility in an alternate to serve as a default. This bracketed else must contain some command which halts test case execution.

Any statements which occur in sequence after the last alternate in an `alt` construct are immediately suspect and must be commented to precisely indicate why the writer has done so and why the alternative solutions above it cannot apply. Again, avoid them if at all possible.

A.3.13 PICS and PIXIT parameters

TTCN-3 has an elegant way of passing configuration information to the test system via the use of module parameter lists. For example:

```
module Style (
ATSPicsType1 atsPics1,        //PICS reference
ATSPixitType atsPixit1 )  //PIXIT reference
{

/*
 *  This is the definitions part.
 */

control
   {
   }
}
```

Of course, one has the option of including these items as variables or constants within the module itself rather than passing them as parameters into the module.

Annex B: Bibliography

- ETSI ES 201 873-2: "Methods for Testing and Specification (MTS); The Tree and Tabular Combined Notation version 3; Part 2: TTCN-3 Tabular Presentation Format (TFT)".
- ETSI TR 101 873-3: "Methods for Testing and Specification (MTS); The Tree and Tabular Combined Notation version 3; Part 3: TTCN-3 Graphical Presentation Format (GFT)".
- ITU-T Recommendation X.680: "Information technology - Abstract Syntax Notation One (ASN.1): Specification of basic notation".
- IETF RFC 2543: "Session Initiation Protocol (SIP)".

History

Document history		
V1.1.1	January 2002	Publication